Reactive Programming of Cellular Automata

Frédéric Boussinot

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Reactive Programming of Cellular Automata

Frédéric Boussinot

N° 5183

Mai 2004

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--5183--FR+ENG

Reactive Programming of Cellular Automata

Frédéric Boussinot

Thème COM — Systèmes communicants
Projet Mimosa

Rapport de recherche n° 5183 — Mai 2004 — 24 pages

Abstract: Implementation of cellular automata using reactive programming gives a way to code cell behaviors in an abstract and modular way. Multiprocessing also becomes possible. The paper describes the implementation of cellular automata with the reactive programming language LOFT, a thread-based extension of C. Self replicating loops considered in artificial life are coded to show the interest of the approach.

Key-words: Reactive programming, Thread, Cellular automata, Artificial life

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Programmation réactive des automates cellulaires

Résumé : La programmation réactive permet d'implémenter les automates cellulaires de manière abstraite et modulaire. Elle rend aussi possible l'utilisation de machines multi-processeurs. Le papier décrit l'implémentation d'automates cellulaires à l'aide du langage LOFT qui est une extension réactive de C, fondée sur la notion de thread. Des exemples des boucles auto-répliquante, dans le domaine de la vie artificielle, sont traités dans ce cadre.

Mots-clés : Programmation réactive, Automate cellulaire, Thread, Vie artificielle

1 Introduction

Cellular automata (CA) are used in various simulation contexts, for example, physical simulations, fire propagation, or artificial life. These simulations basically consider large numbers of small-sized identical components, called *cells*, with local interactions and a global synchronized evolution. Conceptually, evolution of CA is decomposed into couples of steps: during the first step, cells get the states of their neighbors and during the second step they change their own state according to information obtained from previous step. Usually, CA are coded as sequential programs basically made of a single main loop which considers all cells in turn.

Concurrent programming is generally considered to increase modularity as one can often naturally decompose a complex application in several cooperating programs which can be run concurrently. The reactive programming (RP) technique[RP] basically considers software systems made of concurrent components sharing the same global *instants*. Actually, all components are running at the same pace as they necessarily synchronize at the end of each instant. The existence of instants allows one to define *events* which are broadcast to all concurrent components and which can be used for communication or synchronization purposes. A key characteristic of RP is that new concurrent components and new events can be introduced dynamically during system execution. RP has been used for simulations in physics[Sam].

This paper considers the use of RP for implementing CA. The following points are more particularly considered:

- Modularity of programming. The behaviors of cells is rather opaque in usual CA implementations. This is generally not felt as a big issue because cells behaviors are often very simple. However, in some contexts, for example artificial life, one may ask for more complex cell behaviors. In these cases, modularity is required.
- Multiprocessing. Sequential programming of CA makes difficult the use of several processors because cells must be protected from concurrent accesses of their neighbors and because the global synchronization of cells has to be preserved.

The structure of the paper is the following: the language LOFT, based on RP, is described in section 2. CA implementation in LOFT is considered in section 3. Conway's Game of Life and standard self-replicating loops of Langton and Sayama are considered in section 4. A new approach to self-replicating loops is finally proposed in section 5.

2 LOFT

LOFT is a new thread-based concurrent language for reactive programming[RP].The underlying computing model is first described, then the syntax of LOFT is overviewed. Actually, only the constructs that are needed for the implementation of CA are considered.

2.1 Model

Basically, LOFT considers *fair threads*, *schedulers*, and *events*. Fair threads (often simply called threads) are the basic units of concurrency. They are created as instances of *modules* and are under the control of the scheduler to which they are linked. More precisely, all the threads linked to a same scheduler run in cooperation and share a logical clock with which they cyclically synchronize. Time intervals between two ticks of the logical clock are called *instants*.

Because they share instants, threads linked to the same scheduler can synchronize and communicate using *broadcast events*. Events are data which are either present or absent during an instant. Events are non persistent: they are automatically reset at the beginning of each instant. The key point is that, at each instant, all the threads linked to the same scheduler see events presence or absence exactly in the same way: it is not possible for one thread to see an event as present while another one sees it absent during the same instant.

There are two basic instructions related to events: the generation of an event, which make it present for the current instant, and the waiting for an event which blocks execution of a thread until the event is generated. Data can be associated to event generations. As several generations of the same event can occur during the same instant, there is an instruction to get the various data that are generated.

Some threads can be autonomous and not linked to any scheduler. In this case, these threads are only under the control of the OS and are run in a preemptive way. Unlinked threads do not share any logical clock and, unlike linked threads, are thus not able to communicate through broadcast events.

Several schedulers can coexist in the same application defining several *synchronized areas* running at their own pace. LOFT defines primitives for threads to dynamically unlink from a scheduler and to link to another one.

At the implementation level, all threads and schedulers running in the same application are executed in the same process. Schedulers and threads that have the ability to be unlinked are mapped to *native* kernel threads (actually POSIX Pthreads[NBJ96]), while threads that are always linked are executed by the kernel thread of the scheduler to which they are linked.

2.2 Syntax

Modules and Threads

Modules are the basic units of the language. The syntax is based on C and modules are defined in files that can also contain standard C code. For example, here is the definition of a module named `trace` which at each instant prints its parameter `s`:

```
module trace (char *s)
while (1) do
   printf ("%s\n",local(s));
   cooperate;
end
```

```
end module
```

Parameters are a special case of local variables, which preserve their values between instants. In LOFT, every access to a local variable `v` has the form `local(v)`. Note the syntax of the `while` loop specific to LOFT which differs from the one of the `while` loop of C. The `cooperate` instruction is considered bellow.

Threads are instances of modules. Instances of a module `m` are created using the function `m_create` which returns a new thread running the code of `m`. Threads are of the (pointer) type `thread_t`. The executing thread is returned by the C function `self` and its local data by the function `local_data`.

Instructions

The body of a module is basically a sequence of instructions executed by threads which are created as instances of the module. There are two types of instructions: *atomic* instructions which are run in one single step, and *non-atomic* ones, the execution of which can take several instants to complete.

Atomic instructions are C function calls (like the call of the `printf` function, in the previous module), or C blocks of the form `{...}`. The `generate` C function is used to generate an event (which is of type `event_t`). Thus, calls to `generate` are atomic instructions. For valued generations, the `generate_value` C function is used instead of `generate`.

The `cooperate` instruction is basically non-atomic as it takes two instants to complete: the execution of the executing thread blocks at the first instant and the control is returned to the scheduler; the `cooperate` instruction terminates at the next instant, when the executing thread receives the control again from the scheduler.

The instruction `await` is a non-atomic instruction which completes when the awaited event is generated; completion can occur in several instants in the future (or may be never, in which case the thread blocks forever).

The instruction `get_value` is the third non-atomic instruction. It is the mean to get the values associated to an event by calls of `generate_value`. The values are indexed and accessed one by one. The instruction `get_value (e,n,r)` is an attempt to get the value of index `n` generated for the event `e` during the current instant. If available, the value is assigned to `r` during the current instant, otherwise, `r` will be set to `NULL` at the next instant. In the first case, the function `return_code ()` returns the value `OK` while `ENEXT` is returned when a value was not available. For example, the following module awaits an event and prints all the (integer) values generated with it:

```
module print_all_values (event_t evt)
local int i,int res;
await (local(evt));
{local(i) = 0;}
while (1) do
   get_value (local(evt),local(i),(void**)&local(res));
   if (return_code () == OK) then
```

```
      {
         printf ("value #%d: %d\n", local(i), local(res));
         local(i)++;
      }
   else
      return;
   end
end
end module
```

The second line, starting by the `local` keyword, declares two local integer variables `i` and `res`, whose values are preserved between instants; `i` stores the index of the next value to read and `res` stores the value read.

Schedulers

Schedulers are of type `scheduler_t` and are created by the `scheduler_create` C function. An implicit scheduler is automatically declared and started in every LOFT program. It is the default scheduler for all creations of threads and events. To create a thread instance of a module `m` in a specific scheduler `s`, one uses the function `m_create_in` instead of `m_create`. The current scheduler is returned by the `current_scheduler` C function.

2.3 Summary

The main characteristics of LOFT are:

- It is a concurrent language, based on C and compatible with it.
- Programs can benefit from multiprocessor machines. Indeed, schedulers and unlinked threads can be run in real parallelism, on distinct processors.
- There exist *instants* shared by all the threads linked to the same scheduler. Thus, all threads linked to the same scheduler execute at the same pace, and there is an automatic synchronization at the end of each instant.
- *Events* can be defined by users. They are instantaneously broadcast to all the threads linked to a scheduler; events are a modular and powerful means for threads to synchronize and communicate.

LOFT is strongly linked to an API of threads called FairThreads[Bou01]: actually, LOFT stands for *Language Over Fair Threads*. This API mixes threads with the reactive approach, by introducing instants and broadcast events in the context of threads. LOFT is implemented in a very direct way by a translation into FairThreads.

3 CA Implementation

One considers an implementation of CA in which cells are threads. Cells are not systematically created at system initialization, but only when needed. Moreover, the cell space can be divided into several areas, corresponding to distinct schedulers, for being able to benefit from multiprocessor architectures. These areas synchronize at each instant to maintain the notion of a global instant.

3.1 Cells

The implementation of cells is based on the following points:

- Each cell is a fair thread, with an associated activation event.
- To activate a neighbor, a cell generates the activation event of the neighbor with an associated information describing its own state.
- At each instant, a cell activates its neighbors, then it collects the information from the neighborhood, and finally, it updates its state according to the received information.

In many CA, there is the distinction between *active* and *quiescent* cells. Quiescent cells actually always remain quiescent while their neighborhood is only composed of quiescent cells (empty neighborhood). Implementations should consider this and avoid any processing of quiescent cells. A cell cyclically performs the following sequence of actions:

1. if the cell is quiescent, await an activation from the neighborhood then proceed to action 2;
2. activate all the cells in the neighborhood and communicate them the cell state;
3. collect the states of all the cells in the neighborhood;
4. change the cell state according to the information previously got.

Of course, for a quiescent cell the first action must avoid "busy-waiting" while the neighborhood remains empty. Note that only active cells and cells belonging to the neighborhood of an active cell are activated. The following module implements cells:

```
module cell (int x,int y,event_t activation,int status,int state)
local neighborhood_t neighborhood, info_t info,int count,int more;
initialize_cell (self ());
while (1) do
     if (local(status) == QUIESCENT) then
       await (local(activation));
     else
       activate_neighborhood (self ());
```

```
     end
     {
         local(more) = 1;
         local(count) = 0;
         clear_neighborhood (local(neighborhood));
     }
     while (local(more)) do
        get_value (local(activation),local(count),(void**)&local(info));
        {
           if (return_code () == OK) {
              set_neighbor (local(neighborhood),local(info));
              local(count)++;
           } else local(more) = 0;
        }
     end
     cell_behavior (self ());
end
end module
```

The local variable `status` defines the cell status (quiescent or not) and the local variable `state` defines the cell state. The two types `neighborhood_t` and `info_t` are auxiliary types used to store cell information. The function `cell_behavior` implements the specific behavior of the cell, which defines how the state and the status of the cell change according to the neighbors.

Note that no explicit cooperation (`cooperate` instruction) is needed because an implicit cooperation is actually embedded with the `get_value` instruction: the internal `while` loop is exited only at the next instant, as the loop body always waits for a new information during the whole current instant. Thus, `return_code` returns the value `ENEXT` at the next instant, which makes the loop exit only at that time.

3.2 Neighborhood

The geometry of cells can vary in CA. An important class, which is the one considered here, is the one of 2-dimensional CA where cells are structured as arrays. In this context, `NEIGHBOURS` defines the number of neighbours for each cell, and `context` is the mean to get the cell coordinates of neighbors. For example, the 4 elements neighborhood (called, the *von-Neumann* neighborhood) is defined by:

```
#define NEIGHBOURS 4
#define TOP    0
#define RIGHT  1
#define BOTTOM 2
#define LEFT   3

int context[NEIGHBOURS][2] = {{0,-1},{1,0},{0,1},{-1,0}};
```

For managing neighborhoods, one defines `cell_array` which is an array, with the dimensions `MAXX` et `MAXY` of the visualising applet, where all cells are placed. Note that this is an array of pointers. The array is initialized by null pointers by the function `cellular_automaton_init`.

```
thread_t cell_array[MAXX][MAXY];

void cellular_automaton_init ()
{
   int x,y;
   for (x = 0; x < MAXX; x++)
      for (y = 0; y < MAXY; y++)
        cell_array[x][y] = NULL;
}
```

Cells are created in the current scheduler by the `new_cell` function. It takes as parameters the initial status and state of the created cell. The activation event is also created at that time in the current scheduler.

```
thread_t new_cell (int x,int y,int status,int state)
{
   scheduler_t sched = current_scheduler ();
   return cell_create_in (sched,x,y,event_create_in (sched),status,state);
}
```

Neighbors are accessed, and created when needed, by the `get_cell` function:

```
thread_t get_cell (int x,int y,int change[2])
{
   int newx = x+change[0];
   int newy = y+change[1];
   if (!inside_applet (newx,newy)) return NULL;
   if (cell_array[newx][newy] != NULL) return cell_array[newx][newy];
   cell_array[newx][newy] = new_cell (newx,newy,0,0);
   return cell_array[newx][newy];
}
```

Note that a cell is not created if it is not inside the applet (function `inside_applet`). Other choices could have been made as well (a cylinder space, for example).

The function that activates the neighborhood of a cell can now be defined by:

```
void activate_neighborhood (thread_t me)
{
   int dir;
   cell_data_t d = local_data (me);
   for (dir = 0; dir < NEIGHBORS; dir++) {
```

```
      thread_t cell = get_cell (d->x,d->y,context[dir]);
      if (cell == NULL) continue;
      info_t info = make_info (OPPOSITE(dir),d->status,d->state);
      generate_value (d->activation,info)
   }
}
```

The direction from which activation comes is coded in the transmitted information (using the macro `OPPOSITE` which gives the opposite direction of its parameter; for example, `OPPOSITE(RIGHT)` is `LEFT`).

3.3 Firing of cells

In some situations (in section 5, for example) there is the need to *fire* a cell during execution, that is to transmit it an order to change its state and status. The actual change of state and of status is however only of the responsibility of the fired cell, to avoid nondeterminism and race conditions. Firing a cell is done by the following function `fire`:

```
int fire (thread_t source,int target_dir,int new_status,int new_state)
{
   info_t info;
   cell_data_t d = local_data (source);
   thread_t target = get_cell (d->x,d->y,context[target_dir]);
   if (target == NULL) return;
   info = make_info (OPPOSITE(target_dir),d->status,d->state);
   info->fired = 1;
   info->new_status = new_status;
   info->state = new_state;
   generate_value (d->activation,info)
}
```

3.4 Synchronized areas

CA are basically deterministic and cells are sharing global instants. In order to use multiprocessing, one thus needs to synchronize processors in order to preserve determinism and to define common instants. A solution is to define several *synchronized areas*, containing threads linked to the same scheduler, and to synchronize schedulers. This can be done rather easily in LOFT. The advantage is that true parallelism becomes possible for schedulers execution.

Schedulers are placed in an array `area_array` of size `AREA_NUM`. A draw event is associated to each scheduler, which is generated locally when cells need to be drawn. A special module `transfer` is created in each scheduler for transferring the local draw orders to the graphics module which is run by a dedicated scheduler. The transfer of drawing orders to the graphical scheduler is controlled by events of the array `transfer_array`.

```
scheduler_t area_array     [AREA_NUM];
event_t     draw_array     [AREA_NUM];
event_t     transfer_array [AREA_NUM];
```

The initialization of these arrays is made by the function:

```
void controller_init ()
{
  int i;
  for (i = 0; i < AREA_NUM; i++) {
     area_array[i] = scheduler_create ();
     draw_array[i] = event_create_in (area_array[i]);
     transfer_array[i] = event_create_in (area_array[i]);
     transfer_create_in (area_array[i],go_array[i],draw_array[i],draw);
  }
}
```

The controller executes *phases* during which it gives the control in parallel to all the schedulers in `area_array`, and then waits for their completion. When all have returned the control, then a new phase is issued if new activation events have been generated. Otherwise, the schedulers are asked in turn to transfer their draw orders to the graphical scheduler. When all graphical orders have been transfered, the current instant is closed for all schedulers, and the next instant can take place.

For sake of simplicity, the definition of the module `controller` is not given here, and only the module `synchro` which is basic unit for controlling the reaction of one scheduler is considered. Module `synchro` uses the special `scheduler_react` function provided to control schedulers execution. It is a native module because, during reactions, the controlled scheduler should be unlinked in order to be executed by a kernel thread. The module first waits for the triggering event `go`; then it unlinks and a reaction of the controlled scheduler is performed; when the reaction is finished, the module re-links and generates the signaling event `done`. The code is the following:

```
module native synchro (scheduler_t controlled, event_t go, event_t done)
local scheduler_t sched,int kind;
{local(sched) = current_scheduler ();}
while (1) do
    await (local(go));
    get_value (local(go),0,(void**)&local(kind));
    unlink;
    scheduler_react (local(controlled),local(kind));
    link (local(sched));
    generate_value (local(done),NULL);
end
end module
```

Note that the value associated to go is transmitted as a parameter to `scheduler_react`. Note also that a value (actually `NULL`) is associated to `done` to be able to count the number of generation of it, for synchronizing the various schedulers.

An important point is that no explicit lock is needed for protecting data from concurrent accesses from distinct schedulers (of course, locks are present in the implementation of LOFT).

4 Examples

One now considers the well-known Game of Life of Conway and self-replicating structures considered in artificial life.

4.1 GOL

The *Game of Life* (GOL) of Conway[Gar70] uses a 8 elements neighborhood (the *Moore* neighborhood). The living and the death of a cell depends on the number N of living cells present in its neighborhood. More precisely, a dead cell turns to a living cell if N equals 3, and a living cell dies if N is different from 2 or 3. Actually, the living cells are just the active ones. The behavior of GOL cells is defined by the following function:

```
void cell_behavior (thread_t cell)
{
   cell_data_t d = local_data (cell);
   int i, neighbours = 0;
   for (i = 0 ; i < NEIGHBORS; i++)
     if (d->neighborhood[i] && d->neighborhood[i]->status != QUIESCENT)
        neighbours++;
   if (d->status == QUIESCENT && neighbours == 3) {
      d->status = !QUIESCENT; // birth
      draw_cell (cell,WHITE);
   } else if (d->status != QUIESCENT && neighbours != 2 && neighbours != 3) {
      d->status = QUIESCENT; // death
      draw_cell (cell,BLACK);
   }
}
```

A well-known shape is the r-pentomino described on figure 1. It is made of 5 cells, and is placed in the middle position of a GOL CA composed of 200x100 cells.

After 1104 instants, 5318 cells have been created and the CA becomes as shown on figure 2.

4.2 Self Replicating Loops

CA are used in several works on artificial life for studying *self-replicating loops* (SR-loops).

Figure 1: The r-pentomino

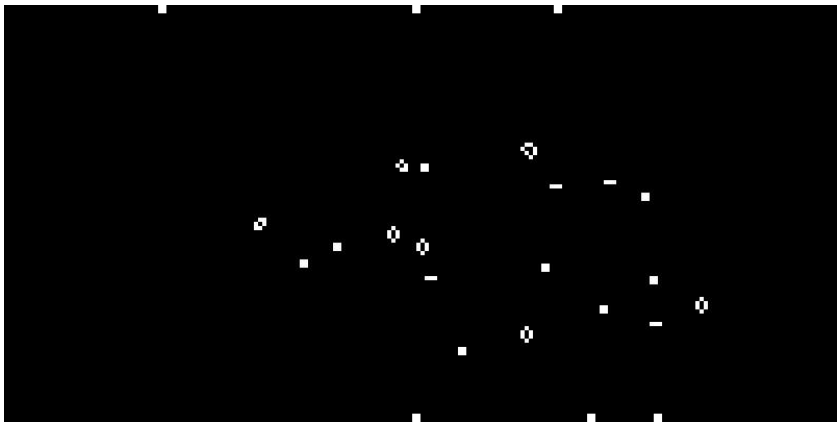

Figure 2: The GOL CA obtained from the r-pentomino

Langton Loop

One of the first SR-loop has been proposed by Langton[Lan84]. It has a state made of 8 different values and is shown on figure 3.

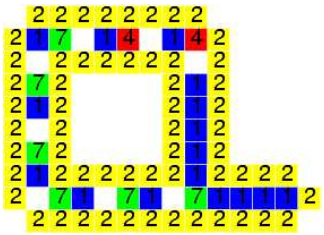

Figure 3: SR-loop

The von Neumann neighborhood is used. The states 7 and 4 are interpreted as "genes" that makes the loop grow and turn when reaching the end of the loop. The cells with state

2 define a sheath shape. The genes travel along the loop counterclockwise and are copied at the T part of the loop. After 3 rotations, when the end of the loop collides with the sheath, the end and the sheath bond together to form two identical loops. Several Java-based applets exist on the Web that show the loop behavior[1].

The behavior of cells is defined by a set of rules stored in a look-up table. Entries in the table are of the form CTRBLN where C is the current state of the cell, T,R,B,L are the states of the top, right, bottom, and left neighbors, and N is the new state of the cell. For example, the rule 122277 codes for the progression of a grow gene at the end of the loop. The cell behavior simply consists in changing the state and the status of the cell according to the look-up table.

Sayama Loops

Sayama[Say98a, Say98b] has defined a variant of the SR-loop, called SDSR-loop, in which a loop destroys itself when colliding with an other loop. Figure 4 is obtained from the SDSR-loop.

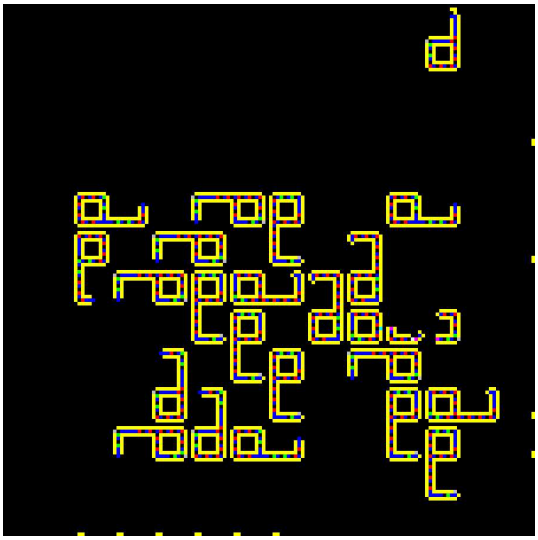

Figure 4: Sayama's SDSR-Loop

The Evo-loop from Sayama[Say99] is a variant of the SDSR-loop which in some colliding situations can change its "genotype" and produce smaller loops. These smaller loops are "more active" and some kind of evolutionary process emerges from the system. The Evo-loop is defined by a specific look-up table with 9^5 elements. In [Say98a], Sayama advocates

[1] See for example `http://cell-auto.com/links/` which lists cellular automata related resources.

for a way to express cell behaviors in a higher-level way than with look-up tables. This is what is proposed in the next section.

5 Programmed SR Loops

One considers self-replicating loops in which behaviors are not described by a look-up table, but are programs. Actually, one considers a very simple loop shown on figure 5, called PSR-loop (Programmed SR-Loop).

Figure 5: PSR-Loop

The PSR-loop does not have any sheath. It is made of one unique sequence of genes, starting with a (red) turn-gene followed by two (green) grow-genes. At the end of the arm is a (blue) creator, for creating new cells.

The *status* of a cell is extended to code, when the cell is not quiescent, a direction from which genes should propagate (TOP, RIGHT, BOTTOM, or LEFT). The *state* of a cell is one of the following:

- BASIC: standard state
- COLLIDE: arm collides
- BARRIER: barrier raised, after collision
- GROW_GENE: gene for growing the arm
- TURN_GENE: gene for turning
- CREATOR: creator of the arm
- SPROUT: gene to create a new arm
- WAITER: wait for a sprout gene, to start arm creation
- STOP: to destroy the arm, after barrier is encountered
- NONE: state of quiescent cells

The `cell_behavior` function is defined, with the help of three auxiliary functions `waiter`, `creator` and `step` defined below, by:

```
void cell_behavior (thread_t cell)
{
   cell_data_t d = local_data (cell);
   if (d->status == QUIESCENT) return;
   if (d->state == WAITER) waiter (cell);
   else if (d->state == CREATOR) creator (cell);
   else step (cell);
   d->image->color = color_of (d->state);
   draw_image (d->image);
}
```

5.1 Step function

The function `step` defines the behavior of standard cells:

- If the cell state is `STOP`, then the cell returns quiescent.
- The `STOP` state propagates back along the arm until the loop body is reached.
- The cell changes its state to `STOP` when a barrier is encountered.
- A pre-waiter is fired when a sprout reaches a corner of the loop.
- The cell state changes to `SPROUT` in front of a barrier.
- The cell state becomes a barrier when a collision is detected.
- The cell state become a sprout when the reverse propagation of `STOP` reaches the loop body.

If none of these actions is applicable, then the state of the previous cell is transmitted to the current cell.

The function `step` is the following:

```
static void step (thread_t cell)
{
   cell_data_t d = local_data (cell);
   int from = FROM(d->status);
   int to = TO(d->status);
   int turn = TURN(from);
   int invturn = INVTURN(from);
   // stop cell disappears
   if (d->state == STOP) {
      d->status = QUIESCENT;
      d->state = NONE;
      return;
   }
```

```
   // reverse progression of stop along the arm
   if (TEST_STATE (to,STOP) && DEAD (turn)) {
      d->state = STOP;
      return;
   }
   // stop if barrier forward
   if (TEST_STATE (to,BARRIER) && d->neighborhood[to]->status != d->status) {
      d->state = STOP;
      return;
   }
   // new waiter created when sprout reaches a corner
   if (TEST_STATE (from,SPROUT) && DEAD (to)) {
      fire (cell,to,from,PRE_WAITER);
      return;
   }
   // barrier produces sprout
   if (TEST_STATE (from,BARRIER)) {
      d->state = SPROUT;
      return;
   }
   // change to barrier state and turn
   if (TEST_STATE (turn,COLLIDE)) {
      d->status = turn;
      d->state = BARRIER;
      return;
   }
   // sprout produced when reverse progression ends
   if (TEST_STATE (to,STOP) && !DEAD (turn)) {
      d->state = SPROUT;
      return;
   }
   // standard state: transmit
   if (d->neighborhood[from]) d->state = d->neighborhood[from]->state;
}
```

The macros `TEST_STATE` and `DEAD` are defined by:

```
#define TEST_STATE(dir,v) (d->neighborhood[dir] && d->neighborhood[dir]->state == v)

#define DEAD(dir) (!d->neighborhood[dir] || d->neighborhood[dir]->status == QUIESCENT)
```

5.2 Waiter function

A waiter is a cell which waits for a turn-gene to become a creator. A pre-waiter cell grows one step and then becomes a waiter. Actually, a pre-waiter is produced when a sprout reaches a corner. In this case, a new arm will be started when a new turn-gene appears. The function `waiter` is defined by:

```
static void waiter (thread_t cell)
{
   cell_data_t d = local_data (cell);
   int from = FROM(d->type);
   int to = TO(d->type);
   // grow one step, and then wait for a turn gene to become a creator
   if (d->state == PRE_WAITER) {
      d->state = BASIC;
      fire (cell,to,from,WAITER);
      return;
   }
   if (d->state == WAITER && TEST_STATE (from,TURN_GENE)) {
      d->state = BASIC;
      fire (cell,to,from,CREATOR);
   }
}
```

5.3 Creator function

A creator is a cell which extends the arm of a loop when reached by a gene. In this case, the creator changes into a basic cell and fires a new creator in the direction specified by the gene. Moreover, a creator detects a collision when its encounters a cell which is not dead; in this case the creator change its state to `COLLIDE`. The function `creator` is:

```
static void creator (thread_t cell)
{
   cell_data_t d = local_data (cell);
   int from = FROM(d->type);
   int to = TO(d->type);
   int turn = TURN(from);
   int invturn = INVTURN(from);
   // collision detected by a creator
   if (!DEAD (to)) {
      d->state = COLLIDE;
      return;
   }
   // genes
   if (TEST_STATE (from,GROW_GENE)) {
      d->state = BASIC;
      fire (cell,to,from,CREATOR);
   } else if (TEST_STATE (from,TURN_GENE)) {
      d->state = BASIC;
      fire (cell,turn,invturn,CREATOR);
   }
}
```

Figure 6 is obtained with PSR-loop (space size 200x200, 1000 instants).

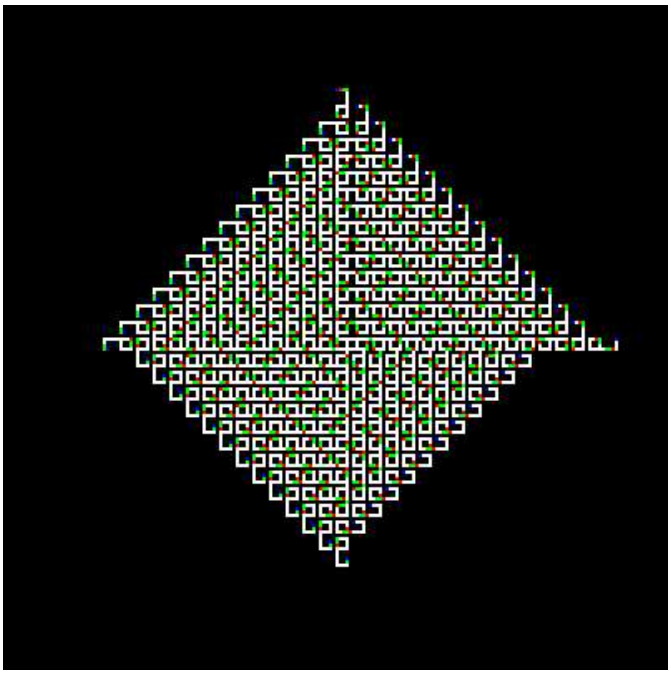

Figure 6: CA from the PSR-Loop

5.4 Self Destruction

One introduces the possibility for a loop to self-destroy when the need for a new cell cannot be satisfied because the cell already exists. A new state `ERASE` is defined: a cell changes its state to `ERASE` when it cannot apply a firing order.

The function `cell_behavior` is extended to deal with the ERASE state:

```
void cell_behavior (thread_t cell)
{
   ...
   // firing of an active cell: destruction of the cell
   if (d->fired && d->type != QUIESCENT) {
      d->state = ERASE;
      draw_cell (cell,color_of (ERASE));
      return;
   }
   if (!d->fired && (TEST_STATE (to,ERASE) || TEST_STATE (from,ERASE) ||
       TEST_STATE (turn,ERASE) || TEST_STATE (invturn,ERASE))) {
      // erase propagates everywhere
```

```
      d->state = ERASE;
      draw_cell (cell,color_of (ERASE));
      return;
   }
   if (d->state == ERASE) {
      d->type = QUIESCENT;
      d->state = NONE;
      draw_cell (cell,color_of (NONE));
      return;
   }
   ...
}
```

The destruction of loops in case of collision is shown on figure 7 (the CA is called PSDSR-Loop; space size is 200x200; 1000 instants are considered).

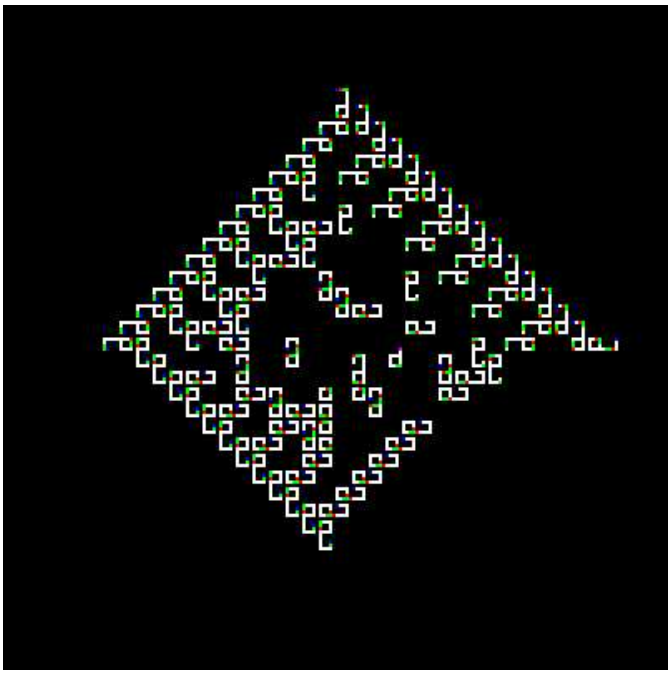

Figure 7: PSDSR-Loop